Deep Learning in Robotics

Neural Networks:
XOR Problem, Multilayer Networks, Backpropagation

Berthold Bäuml

Autonomous Learning Robots Lab

DLR Institute of Robotics and Mechatronics

berthold.baeuml@dlr.de

Organizational

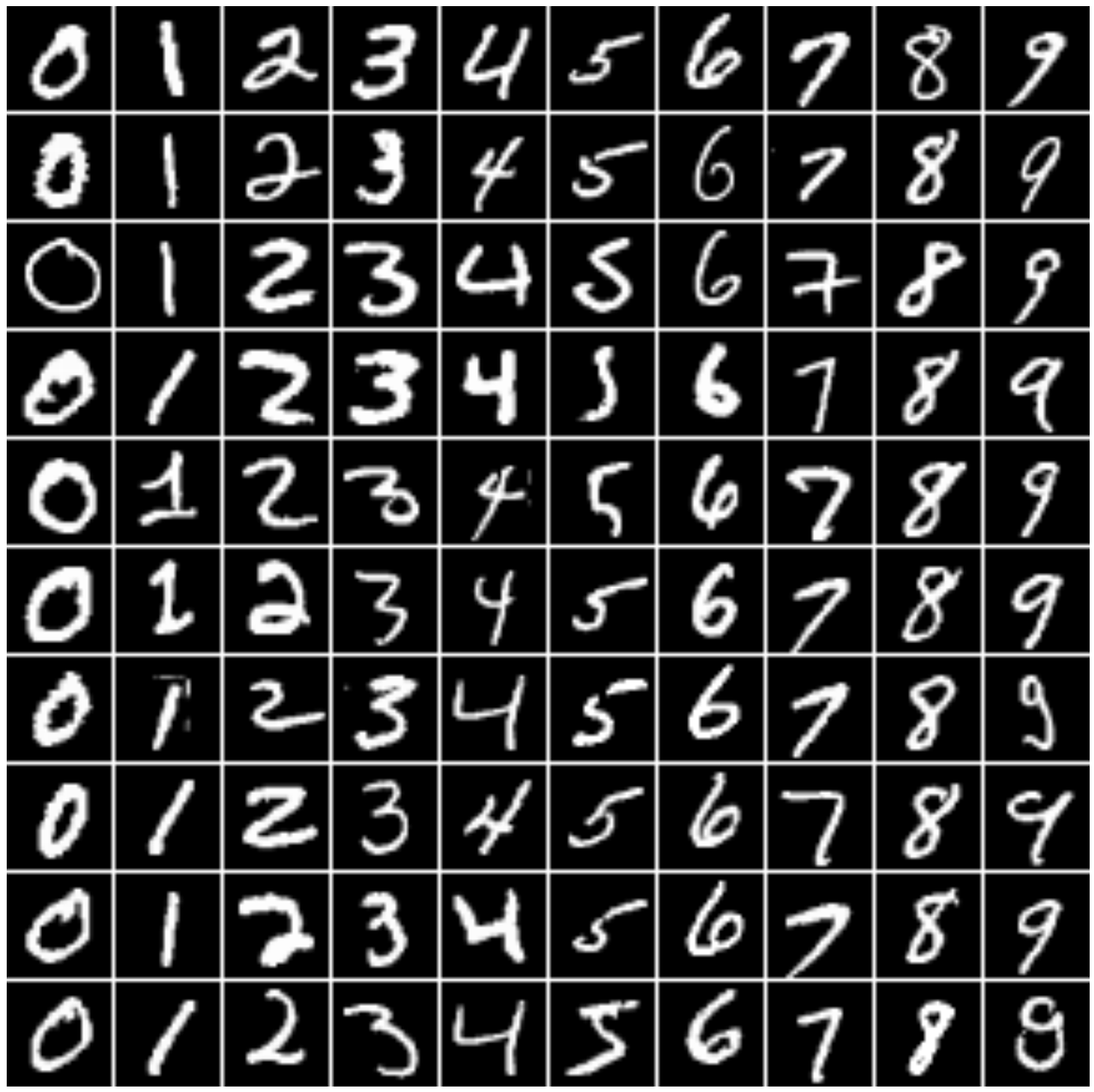
- **approval of lecture only for master's course "Robotics, Cognition, Intelligence"!!!**
- Exams on last lecture date: **27.07.2017**
- please **register asap** so that we know the numbers for room selection

- contact: berthold.baeuml@dlr.de
- information: http://www6.in.tum.de/Main/TeachingSS2017DeepLearning
  - password for lecture slides: TUM17DLR
- important messages by email via TUMOnline

Example: Digit Classification MNIST

MNIST data set
http://yann.lecun.com/exdb/mnist/

- K=10 classes
- Each image 28x28 = 784 dimensional
- N=60000 samples in training set
  10000 samples in test set



Learning algorithm

- Linear logistic regression model
  (no feature mapping)
- Maximum likelihood error function
- Stochastic Gradient Descent with
  learning rate: 0.13
  batch size: 600
  weights initialized with 0

Result

- **error rate: 7.5%**
- currently best error rate: 0.21%
  (deep neural network)

Multiclass Logistic Regression (K>2)

- K classes C_k with 1-of-K coding scheme for target variable $\mathbf{t}$, e.g., $\mathbf{t} = (0, 0, 1, 0, 0)^{\mathrm{T}}$.

- For each class C_k a regression model with "activation function"

  $$a_k = a(\mathbf{x}, \mathbf{w}_k) = \mathbf{w}_k^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}).$$

- Conditional distribution for class C_k by the **softmax** transformation and distribution for $\mathbf{t}$

  $$p(C_k|\mathbf{x}, \mathbf{w}_k) = y_k = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$$

  $$p(\mathbf{t}|\mathbf{x}, \mathbf{W}) = \prod_k^K y_k^{t_k},$$

  with weight matrix $\mathbf{W} = (\mathbf{w}_1, \ldots, \mathbf{w}_K)$.

- The discriminant function

  $$f(\mathbf{x}) = \arg\max_{\mathbf{t}} p(\mathbf{t}|x).$$

Multiclass Logistic Regression (K>2)

- The posterior given data set $(\mathbf{T}, \mathbf{X}) = \{(\mathbf{t}_n, \mathbf{x}_n)\}_{n=1,\ldots,N}$

$$p(\mathbf{W}|\mathbf{T}, \mathbf{X}) = \frac{p(\mathbf{T}|\mathbf{X}, \mathbf{W})p(\mathbf{W})}{p(\mathbf{T}|\mathbf{X})}$$

- Maximum likelihood solution

$$E_{\mathrm{ML}}(\mathbf{W}) = -\log p(\mathbf{T}|\mathbf{X}, \mathbf{W}) = -\sum_{n}^{N} \log p(\mathbf{t}_n|\mathbf{x}_n, \mathbf{W})$$

$$E_{\mathrm{ML}}(\mathbf{W}) = -\sum_{n}^{N}\sum_{k}^{K} t_{nk} \log y_{nk}$$

$$\nabla_{\mathbf{w}_j} E_{\mathrm{ML}}(\mathbf{W}) = \sum_{n}^{N} (y_{nj} - t_{nj})\boldsymbol{\phi}_n.$$

with

$$\frac{\partial y_k}{\partial a_j} = y_k(I_{kj} - y_j),$$

Logistic Regression — Summary

- Probabilistic discriminative model
- Cost function by ML or MAP
- Optimization by gradient descent

Implementation with NumPy
But before some Vector/Matrix/Tensor Basics

- source code at: http://github.com/bbaeuml/lecture-ss17-deep-learning
- mnist/logistic_regression_np.py/

- A matrix $\mathbf{M}$ is a two dimensional array with elements

  M_{ij},

  where i is called the row index and j the column index.
  The graphical representation of a $N_1 \times N_2$ matrix is

$$\mathbf{M} = \begin{pmatrix} M_{11} & M_{12} & M_{13} & \dots & M_{1N_2} \\ M_{21} & M_{22} & M_{23} & \dots & M_{2N_2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M_{N_1 1} & M_{N_1 2} & M_{N_1 3} & \dots & M_{N_1 N_2} \end{pmatrix}$$

- A vector $\mathbf{v}$ is a "one dimensional" matrix with elements

  v_i,

  with dimension N and graphical representation

$$\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{pmatrix}$$

  Hence, we have column vectors meaning that $\mathbf{v}$ could be interpreted as a matrix with $N_2 = 1$ column and $N_1 = N$ rows.

- Transpose of a matrix

$$\left(\mathbf{M}^{\mathrm{T}}\right)_{ij} = M_{ji}$$

$$\mathbf{M}^{\mathrm{T}} = \begin{pmatrix} M_{11} & M_{21} & M_{31} & \dots & M_{N_2 1} \\ M_{12} & M_{22} & M_{32} & \dots & M_{N_2 2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M_{1N_1} & M_{2N_1} & M_{3N_1} & \dots & M_{N_2 N_1} \end{pmatrix}$$

- Transpose of a column vector is a row vector.

$$\mathbf{v}^{\mathrm{T}} = \begin{pmatrix} v_1, & v_2, & \dots, & v_N \end{pmatrix}$$

- Matrix product

$$\mathbf{C} = \mathbf{AB}$$

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

- Matrix vector product

$$\mathbf{b} = \mathbf{Ma}$$

$$b_i = \sum_k M_{ik} a_k$$

- A Tensor of rank K is kind of a multi-dimensional array with K dimensions or axes. The elements of a tensor $\mathbf{A}$ are

  $A_{i_1 i_2 \ldots i_K}$,

  with indices $i_k = 1, \ldots, N_k$ and $k = 1, \ldots, K$.

- Transpose of a tensor. Elements of the transposed tensor $\mathbf{A}^{\mathrm{T}}$ are

$$\left(\mathbf{A}^{\mathrm{T}}\right)_{i_1 \ldots i_K} = A_{i_K \ldots i_1}.$$

- Generalized matrix product of tensors $A_{i_1 \ldots i_{K_A}}$ and $B_{j_1 \ldots j_{K_B}}$,

$$C_{i_1 \ldots i_{K_A - 1} j_1 \ldots j_{K_B - 2} j_{K_B}} = \sum_k A_{i_1 \ldots i_{K_A - 1} k} B_{j_1 \ldots k j_{K_B}}.$$

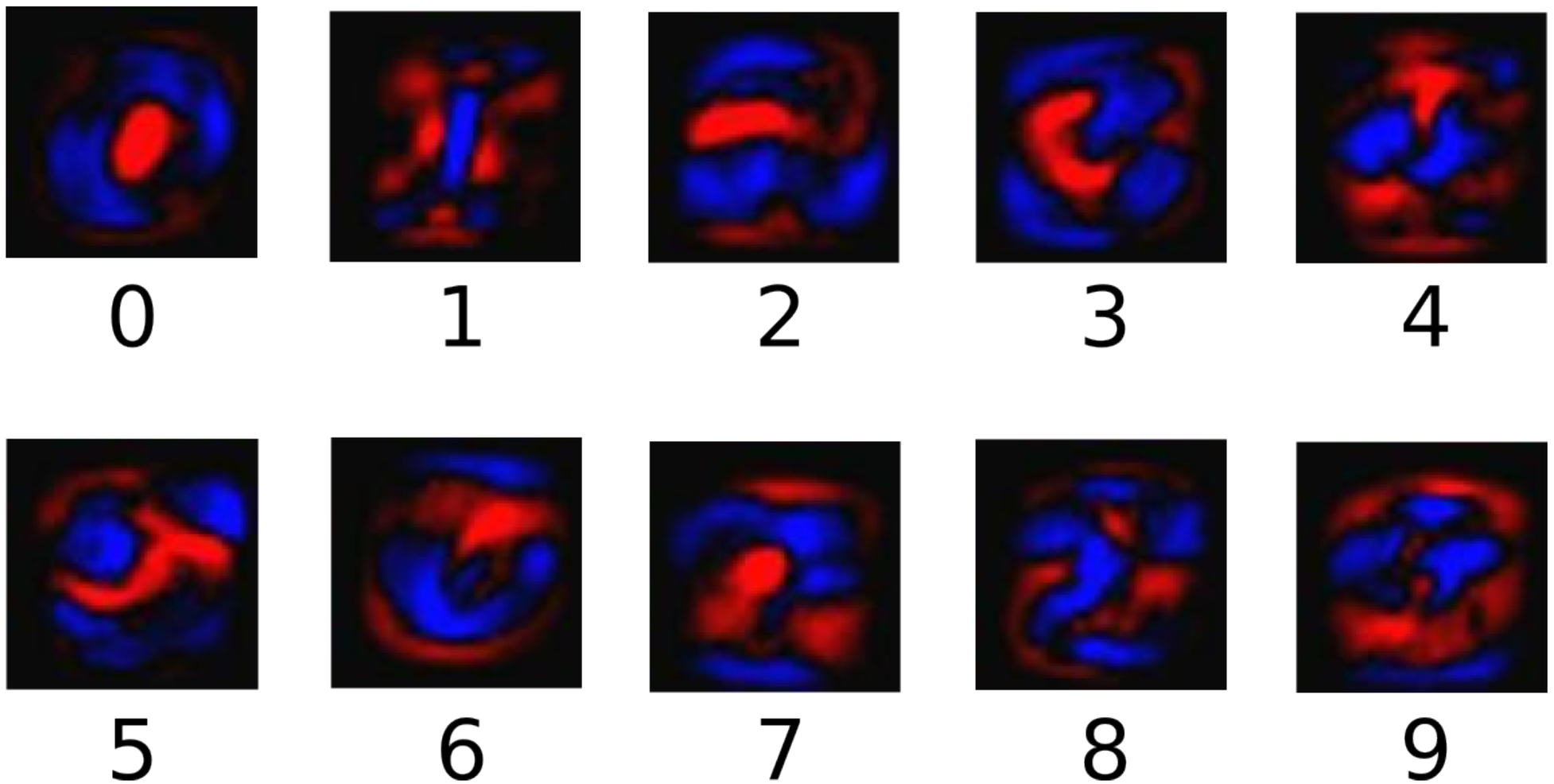
- Tensor product of two tensors $\mathbf{A}$ with rank K_A and $\mathbf{B}$ with K_B over L axes pairs $\{(a_1, b_1), \ldots, (a_L, b_L)\}$ results in a tensor $\mathbf{C}$ with rank $K_C = K_A \times K_B - L$.

$$C_{i_1 \ldots i_{a_1 - 1} i_{a_1 + 1} \ldots i_{a_L - 1} i_{a_L + 1} \ldots i_{K_A} j_1 \ldots j_{b_1 - 1} j_{b_1 + 1} \ldots j_{b_L - 1} j_{b_L + 1} \ldots j_{K_B}} =$$

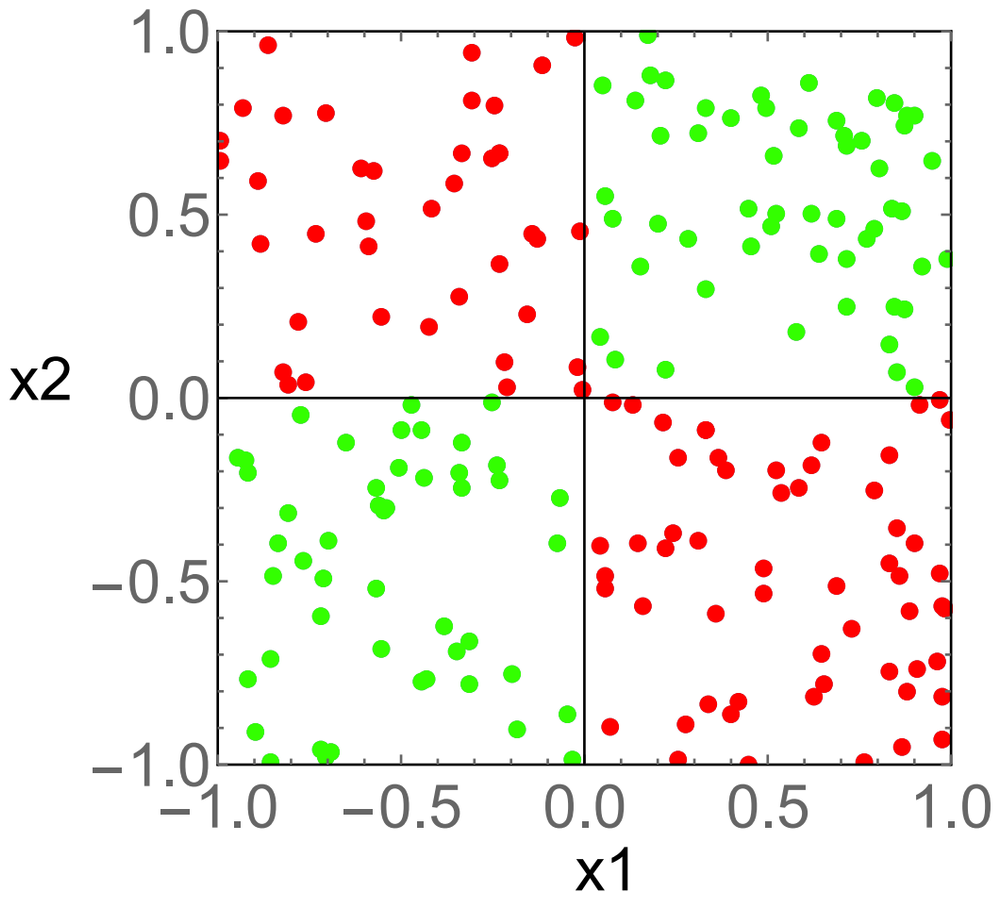
$$= \sum_{k_1, \ldots, k_L} A_{i_1 \ldots i_{a_1 - 1} k_1 i_{a_1 + 1} \ldots i_{a_L - 1} k_L i_{a_L + 1} \ldots i_{K_A}} B_{j_1 \ldots j_{b_1 - 1} k_1 j_{b_1 + 1} \ldots j_{b_L - 1} k_L j_{b_L + 1} \ldots j_{K_B}}$$

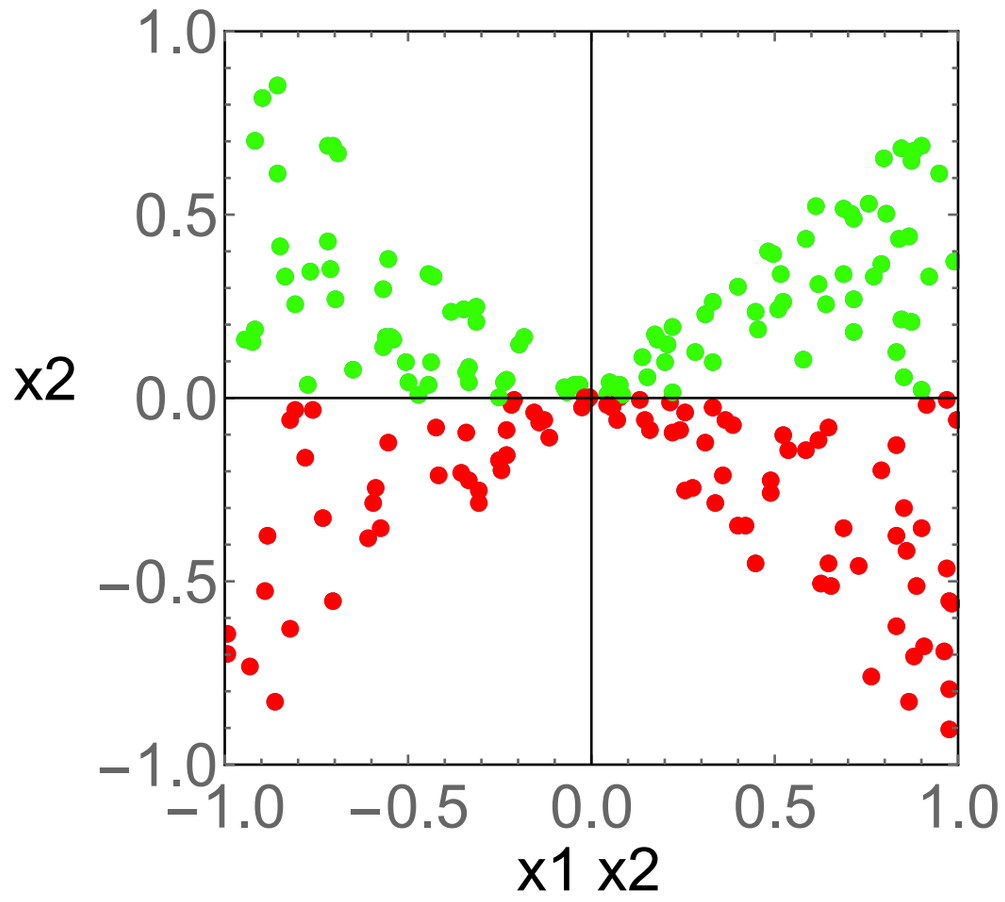
MNIST Logistic Regression
Feature (Weights) Visualization

$$a_k = a(\mathbf{x}, \mathbf{w}_k) = \mathbf{w}_k^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x})$$



XOR-Problem
Classical Approach





- Linear logistic regression for $K = 2$

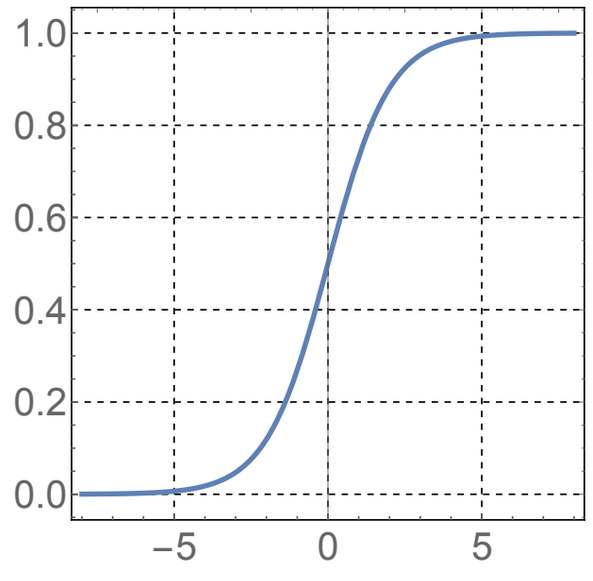
  $\mathbf{y} = \sigma(\mathbf{w}^{\mathrm{T}} \boldsymbol{\phi}(\mathbf{x}))$.

- Not linear separable; linear feature map not working

  $\boldsymbol{\phi}(\mathbf{x}) = (1, x_1, x_2)$

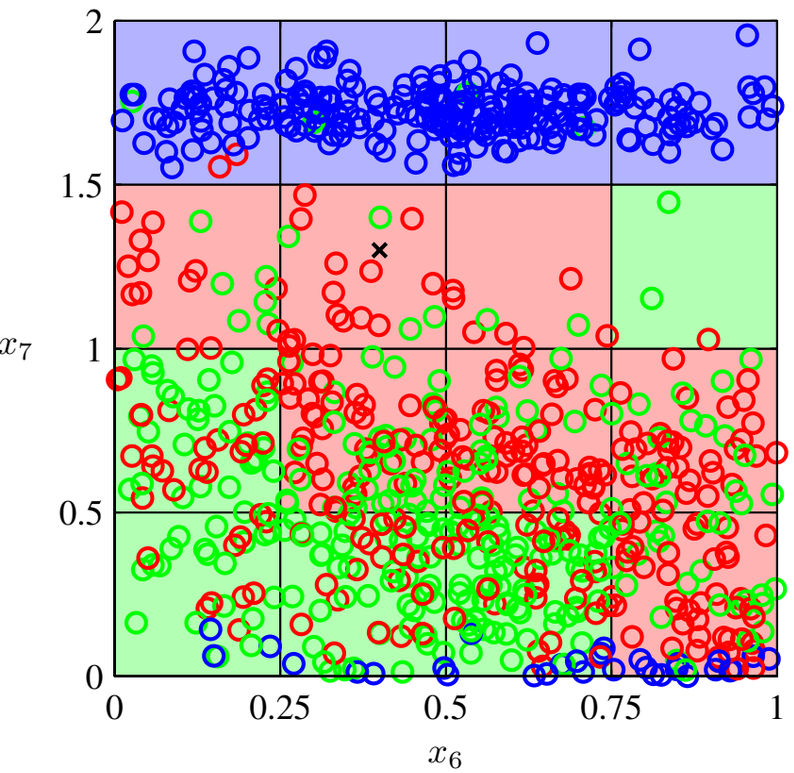
- Classical approach: add new static features!

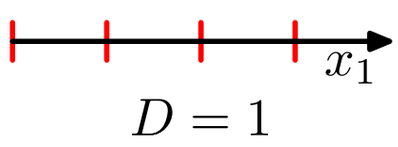
  $\boldsymbol{\phi}(\mathbf{x}) = (1, x_1, x_2, x_1 x_2)$



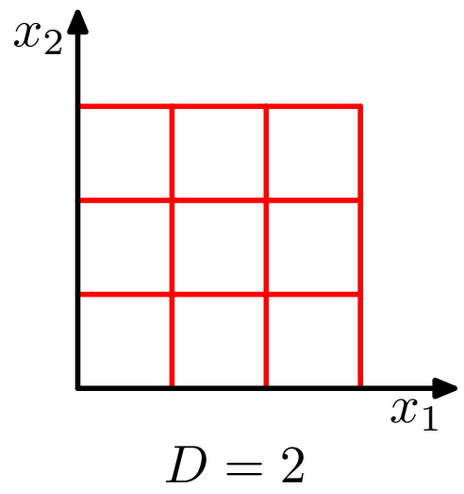
Curse of Dimensionality

- For general, flexible model need many additional features
- Leads in effect to “local interpolation” around training samples for generalization.
- Curse of dimensionality: needed training data to cover whole space increases exponentially with dimension.
- Countermeasure: dimensionality reduction (e.g., with PCA) before feeding into general classifier
- -> non-homogenous learning algorithm

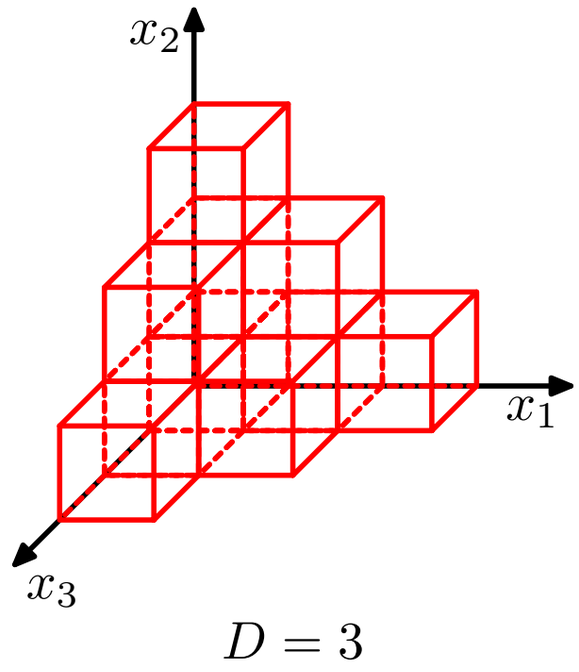




$D = 1$

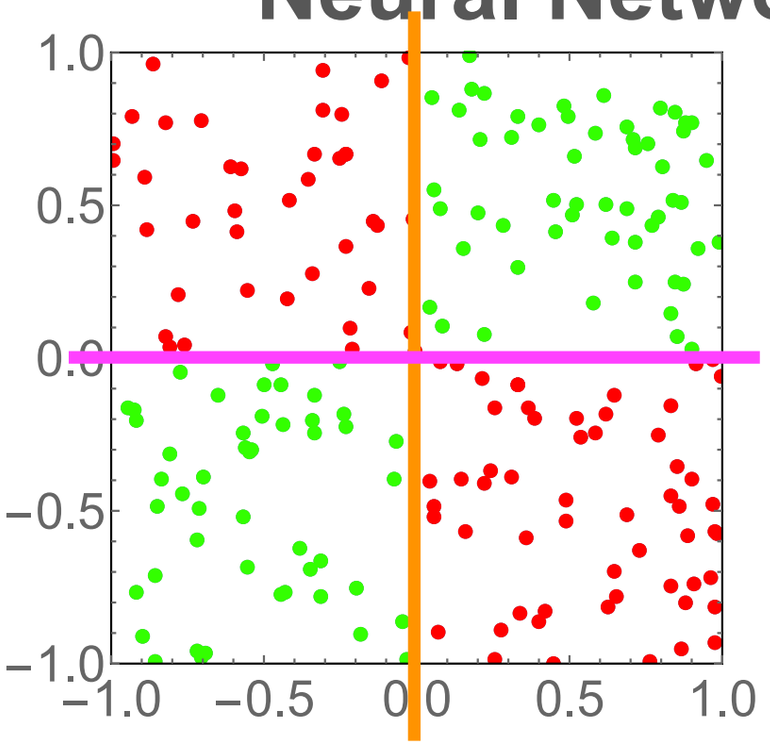


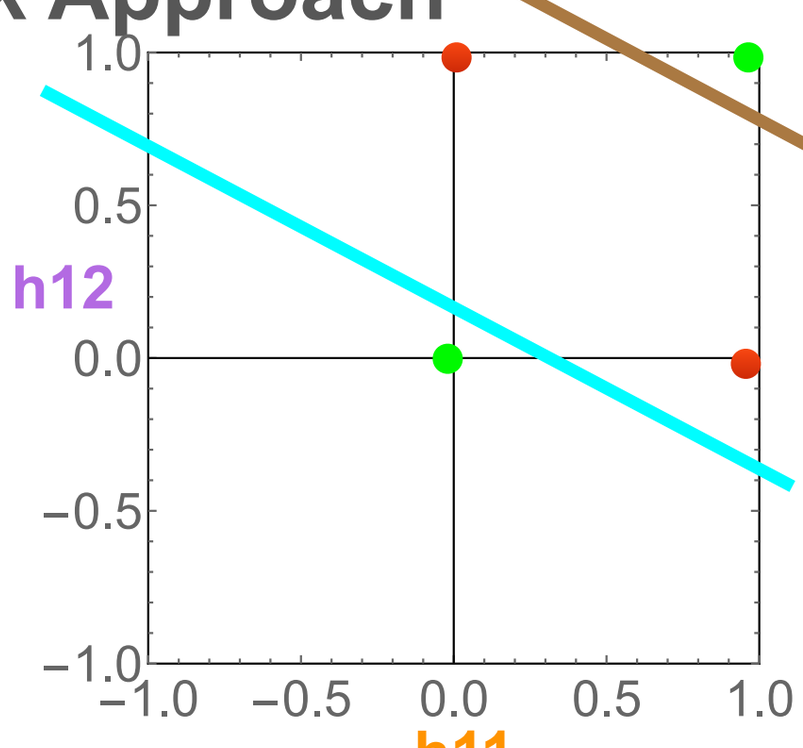
$D = 2$

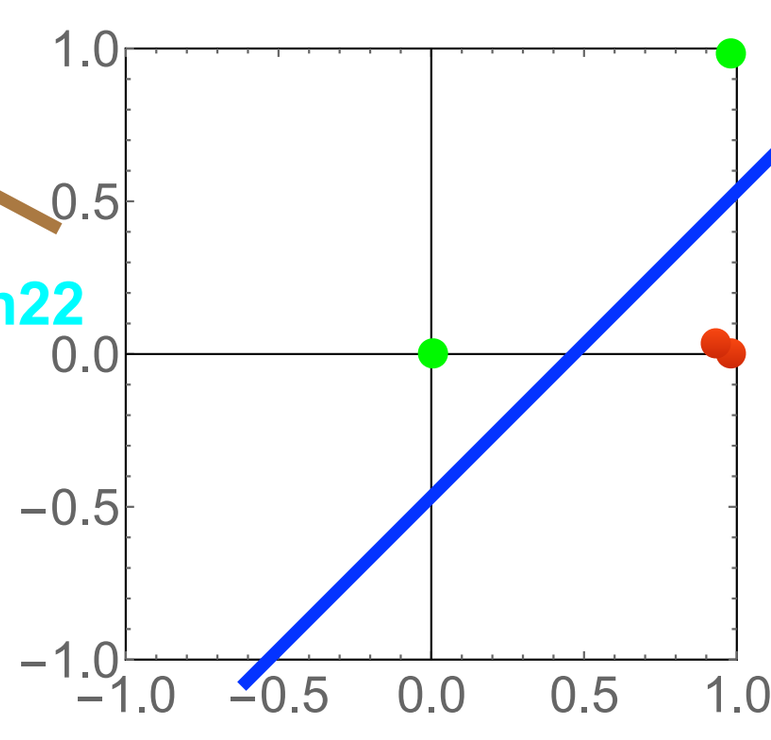


$D = 3$

XOR Problem
Neural Network Approach







- Stacking linear classifiers with linear mapping (weights w and bias b)

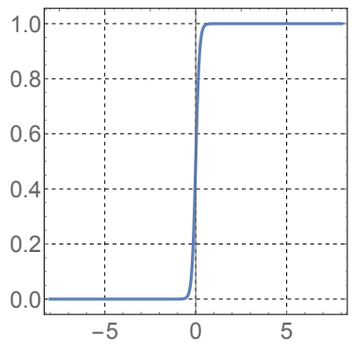
$$\mathbf{y} = \sigma(\mathbf{w}^{\mathrm{T}}\mathbf{x} + b)$$

Here, three layers of classifiers

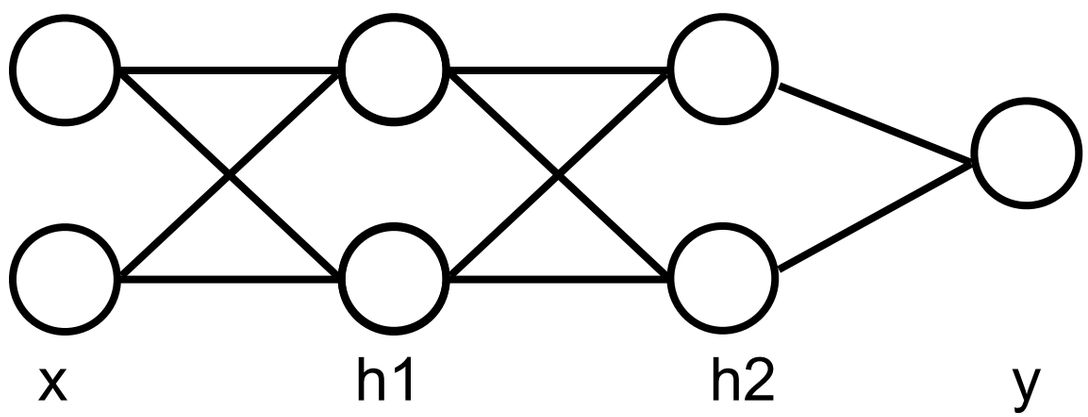
$$\mathbf{h}_1 = \sigma(\mathbf{w}_1{}^{\mathrm{T}}\mathbf{x} + b_1)$$
$$\mathbf{h}_2 = \sigma(\mathbf{w}_2{}^{\mathrm{T}}\mathbf{h}_1 + b_2)$$
$$\mathbf{y} = \sigma(\mathbf{w}_3{}^{\mathrm{T}}\mathbf{h}_1 + b_3)$$

- Interpretation: learning features!

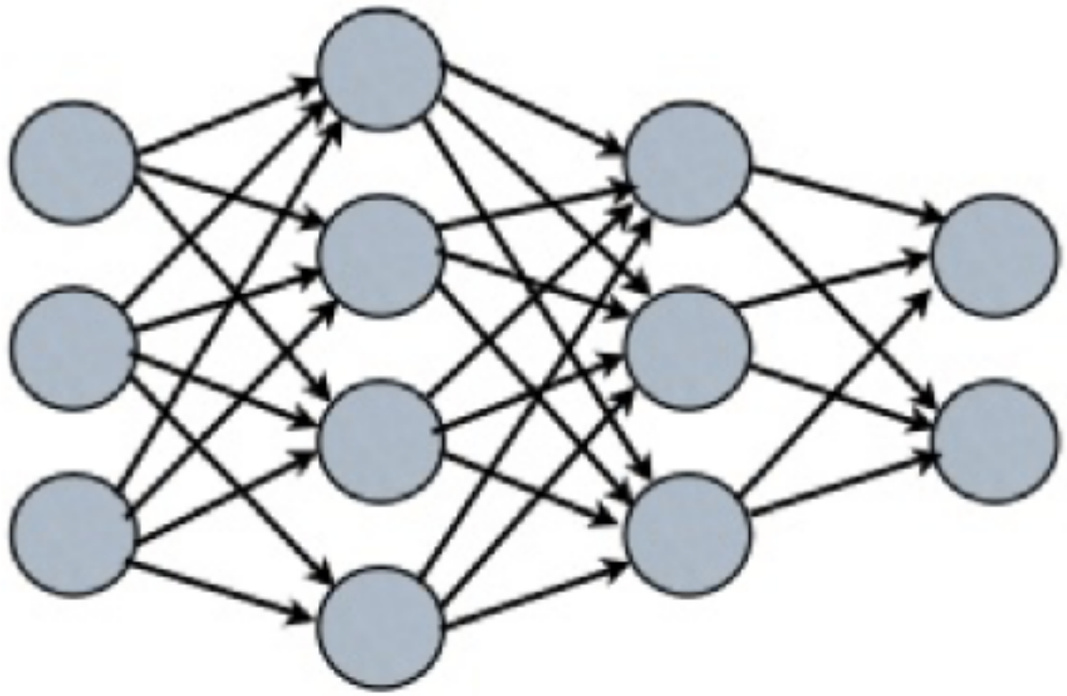
$$\mathbf{y} = \sigma(\mathbf{w}^{\mathrm{T}}\boldsymbol{\phi}(\mathbf{x}, \boldsymbol{\theta}) + b)$$



- Nonlinearity is essential for making "hard decisions".

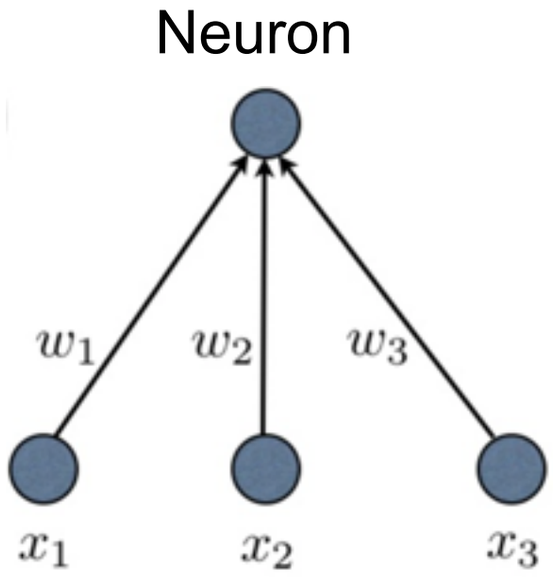


Multilayer Neural Networks



Input Layer **Hidden Layer** **Hidden Layer** **Output Layer**

x **h1** **h2** **y**



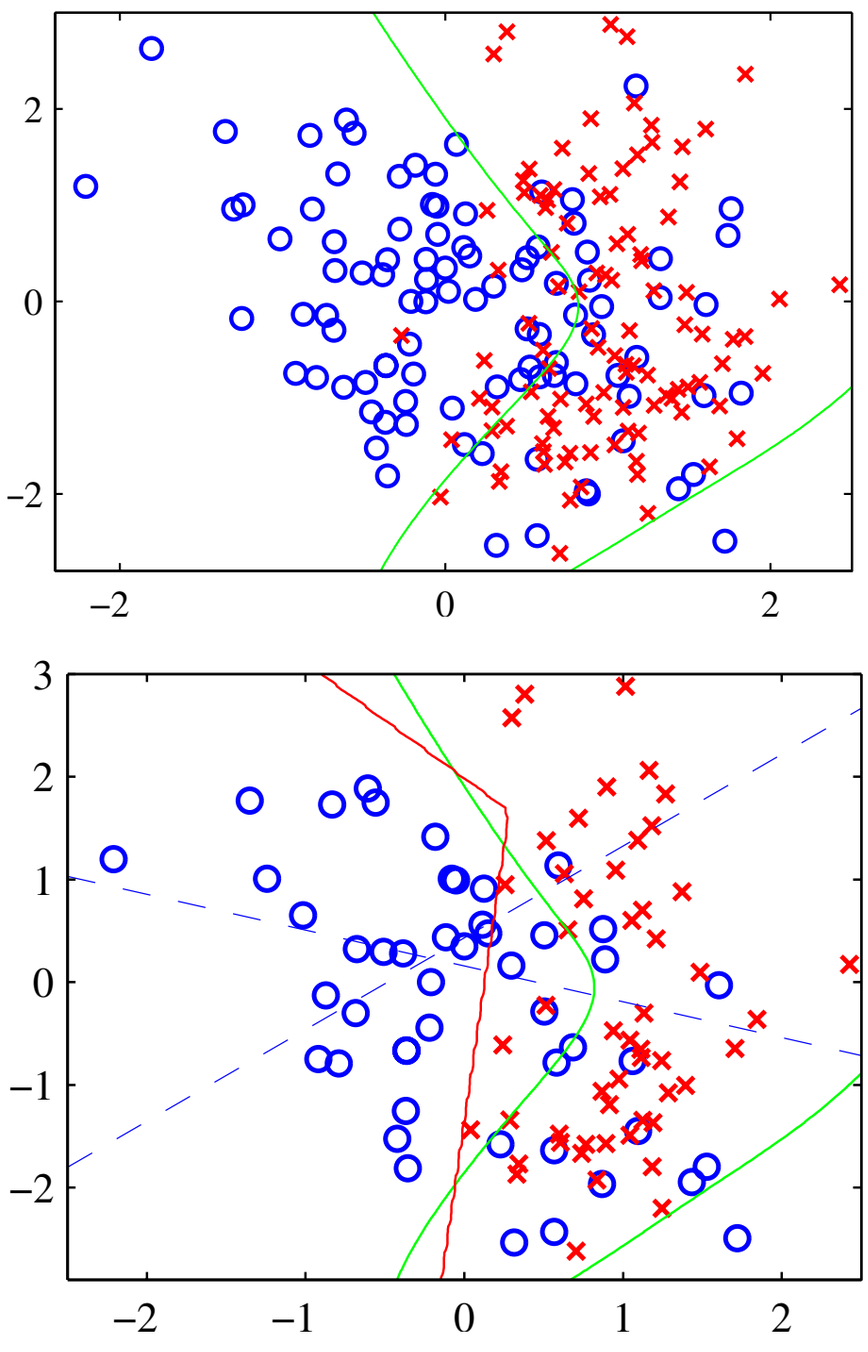
- Neurons/Units compute weighted sum of inputs & apply activation function.

$$h = g(\sum w_i x_i + b)$$

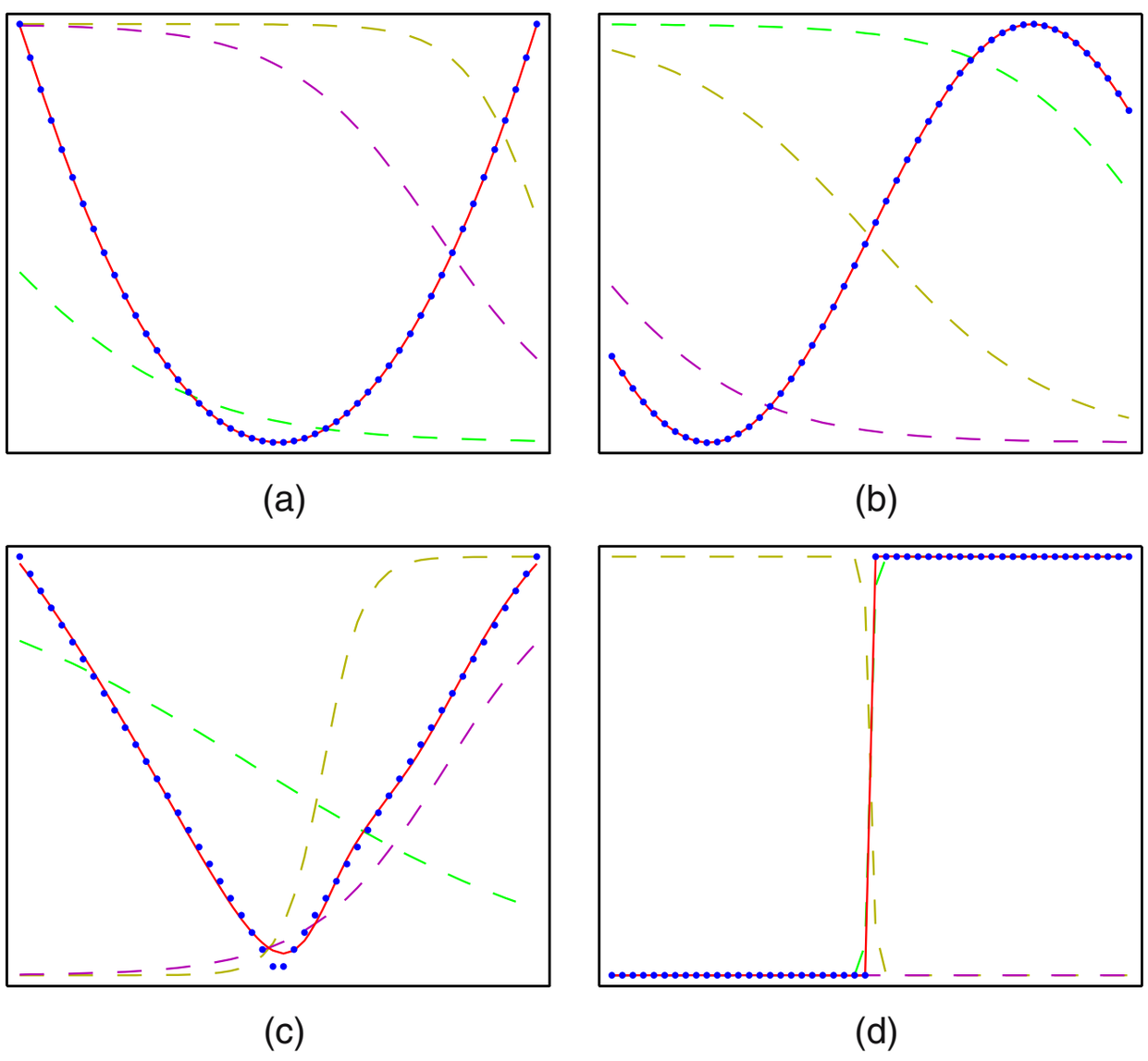
- L-layer neural network with one input, $L-1$ hidden, one output layers.
- Each layer l has N_l identical neurons n with distinct weights and biases.
- Typically, the activation functions of the hidden layers are the same and nonlinear and the one of the output layer is linear.

2-Layer Neural Networks

- Classification: 2 hidden units



- Regression: 3 hidden units



Universal Approximation Theorem

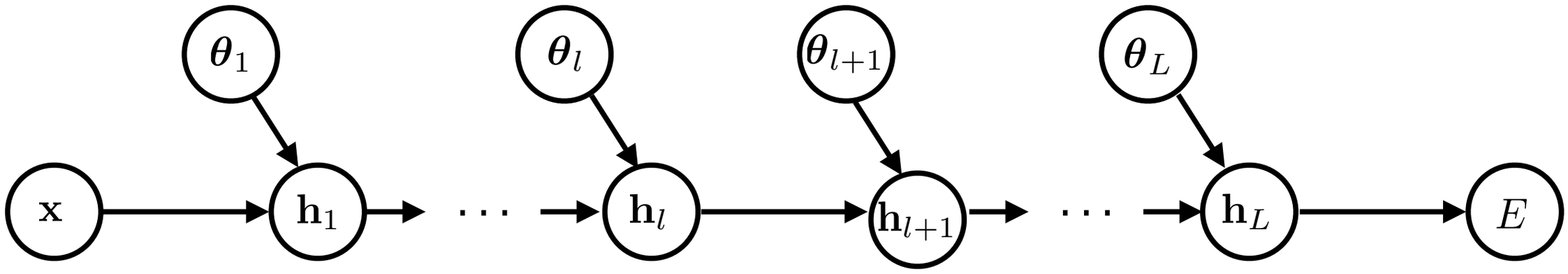
“A two-layer network with linear outputs can uniformly approximate any continuous function on a compact input domain to arbitrary accuracy provided the network has a sufficiently large but finite number of hidden units.”

- Holds for a wide range of hidden unit activation functions

- But theorem does not state:
  - How to find the parameters?
  - How well does it generalize in learning settings?

- Very nice interactive visual proof for sigmoid activation function in http://neuralnetworksanddeeplearning.com/chap4.html

Multilayer Neural Networks



- In vector/tensor notation with all N_l neurons of a layer l combined:

$$\mathbf{h}_l = g_l(\mathbf{W}_l{}^{\mathrm{T}}\mathbf{h}_{l-1} + \mathbf{b}_l), \text{ for } l = 1, \ldots L,$$

with $\mathbf{h}_0 = \mathbf{x}$ and $\mathbf{y} = \mathbf{h}_L$,

with $\mathbf{W}_l$ the $N_{l-1} \times N_l$ dimensional weight matrix, bias vector $\mathbf{b}_l$, and vectorized activation function $g_l(\mathbf{v}) = (g_l(v_1), \ldots, g_l(v_N))^{\mathrm{T}}$ for $\mathbf{v} = (v_1, \ldots, v_N)^{\mathrm{T}}$.

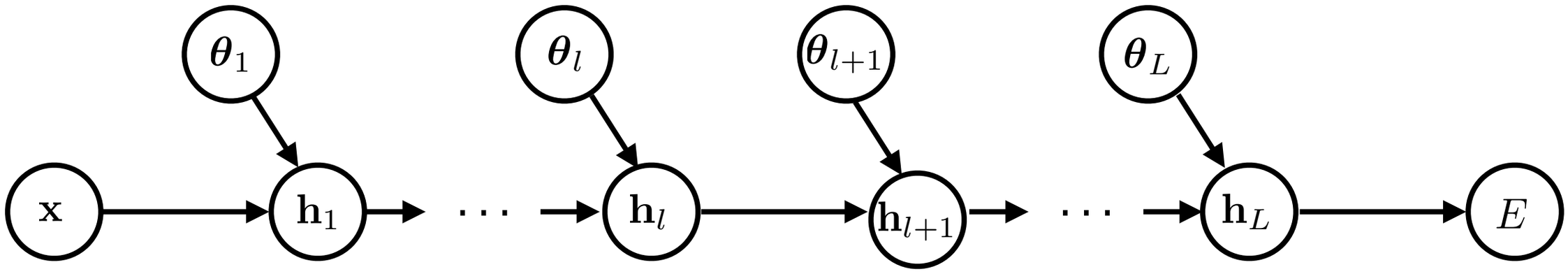
- Output of each layer $\mathbf{h}_l$ depends on the previous layer's output and the model parameters $\boldsymbol{\theta}_l = (\mathbf{W}_l, \mathbf{b}_l)$.

$$\mathbf{h}_l = \mathbf{h}_l(\mathbf{h}_{l-1}, \boldsymbol{\theta}_l)$$

- Training by minimizing error function (e.g., cross entropy) for all parameters $\boldsymbol{\Theta} = (\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_L)$ with gradient descent,

$$E = E(\boldsymbol{\Theta}) = E(\mathbf{y}).$$

Gradients in Multilayer Neural Networks



- Forward sweep: about $O(LN^2)$ operations, assuming $N_l = N$ neurons in each layer.
- Need gradients

  $$\frac{\partial E}{\partial \boldsymbol{\theta}_l}$$

  for all LN parameters $\boldsymbol{\theta}_l$.
- We already know how to compute the "local" gradients

  $$\frac{\partial E}{\partial \mathbf{h}_L}, \frac{\partial \mathbf{h}_{l+1}}{\partial \mathbf{h}_l}, \frac{\partial \mathbf{h}_l}{\partial \boldsymbol{\theta}_l}.$$

- Need to compute gradients along the network layers.

Basics of Derivatives of Higher Dimensional Functions

- Higher dimensional function

$$\mathbf{z} = \mathbf{z}(\mathbf{y}), \qquad \mathbf{y} = \mathbf{y}(\mathbf{x}), \qquad \mathbf{z} = \mathbf{z}(\mathbf{x}) = \mathbf{z}(\mathbf{y}(\mathbf{x}))$$

- Jacobian matrix

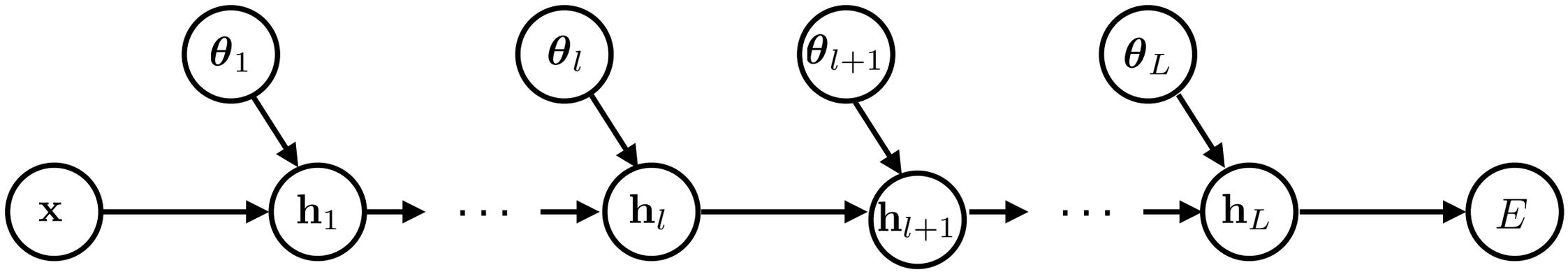
$$\mathbf{J} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}}, \qquad J_{ij} = \frac{\partial y_i}{\partial x_j}$$

- Chain rule

$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \frac{\partial \mathbf{z}}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{x}}, \qquad \frac{\partial z_i}{\partial x_j} = \sum_k \frac{\partial z_i}{\partial y_k} \frac{\partial y_k}{\partial x_j}$$

- Generalizes to matrix and tensor functions.

Gradients in Multilayer Neural Networks



- Numerical differentiation

$$\frac{\partial E}{\partial \theta_{ln}} \approx \frac{E(\boldsymbol{\Theta} + \boldsymbol{\epsilon}_{ln}) - E(\boldsymbol{\Theta})}{\epsilon}, \qquad n = 1, \ldots N_l.$$

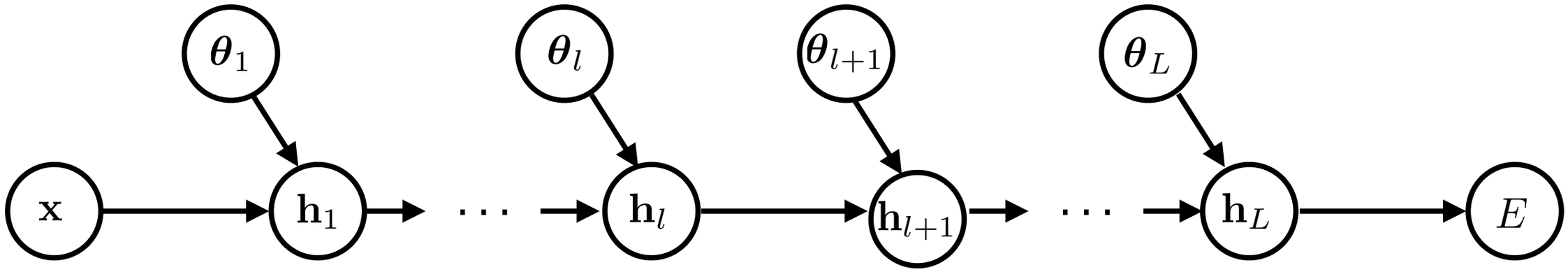
- Forward sweep: $O(LN^2)$ operations
  - for each of the LN^2 parameters one forward sweep.
  - $O(LN^2 LN^2)$ operations; slow in deep networks, $L > 10^2, N > 10^6$

- Naive chain rule

$$\frac{\partial E}{\partial \boldsymbol{\theta}_l} = \frac{\partial E}{\partial \mathbf{h}_L} \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_{L-1}} \cdots \frac{\partial \mathbf{h}_{l+2}}{\partial \mathbf{h}_{l+1}} \frac{\partial \mathbf{h}_{l+1}}{\partial \mathbf{h}_l} \frac{\partial \mathbf{h}_l}{\partial \boldsymbol{\theta}_l}$$

  - for each $\boldsymbol{\theta}_l$ sweep back through chain of Jacobian-vector products
  - $O(LLN^2)$ operations

Gradients in Multilayer Neural Networks



- Can we do better?

$$\frac{\partial E}{\partial \boldsymbol{\theta}_l} = \underbrace{\frac{\partial E}{\partial \mathbf{h}_L}\frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_{L-1}}\cdots\frac{\partial \mathbf{h}_{l+2}}{\partial \mathbf{h}_{l+1}}\frac{\partial \mathbf{h}_{l+1}}{\partial \mathbf{h}_l}}_{\frac{\partial E}{\partial \mathbf{h}_l}}\frac{\partial \mathbf{h}_l}{\partial \boldsymbol{\theta}_l}$$

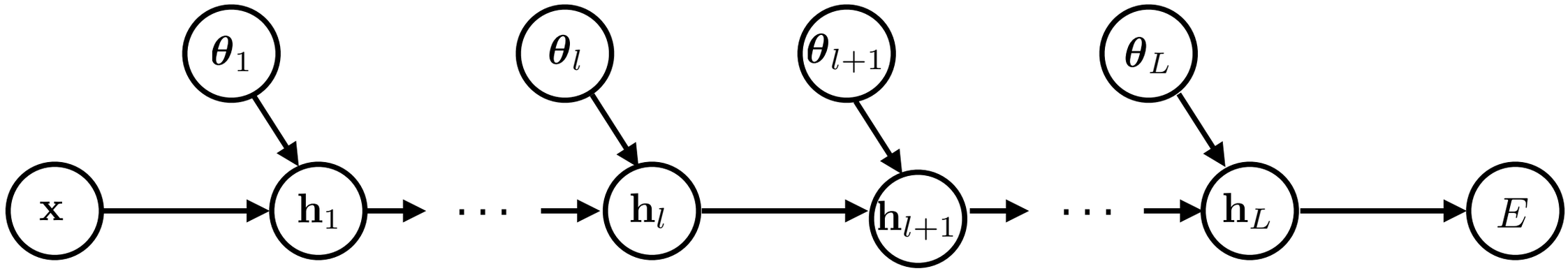
$$\frac{\partial E}{\partial \boldsymbol{\theta}_{l+1}} = \underbrace{\frac{\partial E}{\partial \mathbf{h}_L}\frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_{L-1}}\cdots\frac{\partial \mathbf{h}_{l+2}}{\partial \mathbf{h}_{l+1}}}_{\frac{\partial E}{\partial \mathbf{h}_{l+1}}}\frac{\partial \mathbf{h}_{l+1}}{\partial \boldsymbol{\theta}_{l+1}}$$

- Reuse already computed terms!

$$\frac{\partial E}{\partial \mathbf{h}_l} = \frac{\partial E}{\partial \mathbf{h}_{l+1}}\frac{\partial \mathbf{h}_{l+1}}{\partial \mathbf{h}_l}, \text{ with } l = L, \ldots, 2.$$

$$\frac{\partial E}{\partial \boldsymbol{\theta}_l} = \frac{\partial E}{\partial \mathbf{h}_l}\frac{\partial \mathbf{h}_l}{\partial \boldsymbol{\theta}_l}$$

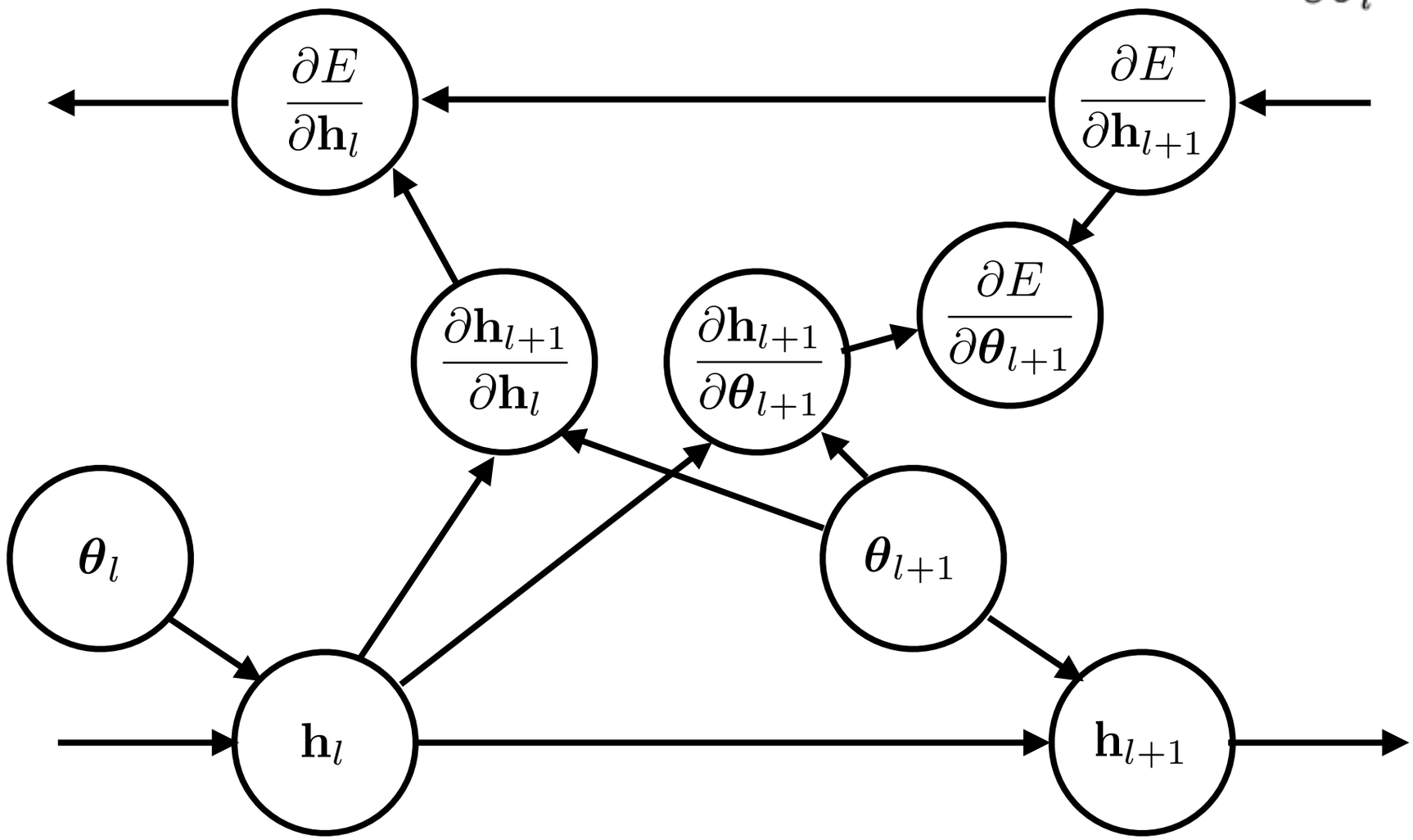
Backpropagation Algorithm!



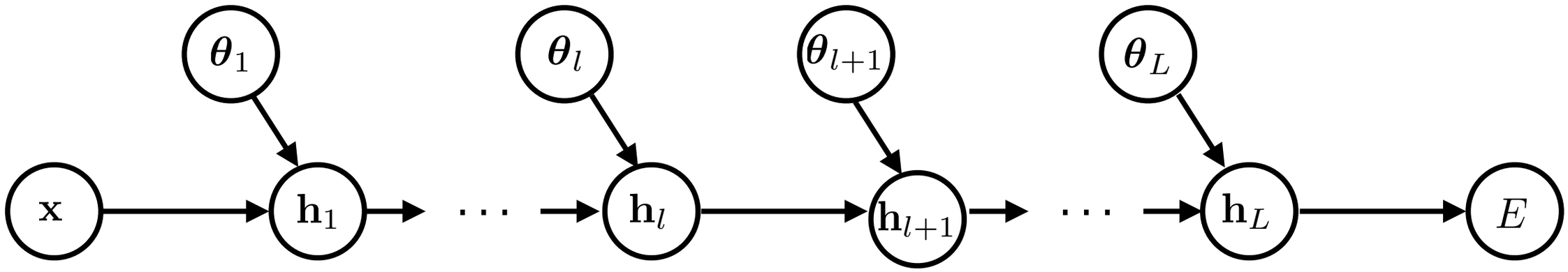
$$\frac{\partial E}{\partial \mathbf{h}_l} = \frac{\partial E}{\partial \mathbf{h}_{l+1}} \frac{\partial \mathbf{h}_{l+1}}{\partial \mathbf{h}_l}$$

$$\frac{\partial E}{\partial \boldsymbol{\theta}_l} = \frac{\partial E}{\partial \mathbf{h}_l} \frac{\partial \mathbf{h}_l}{\partial \boldsymbol{\theta}_l}$$

Adding nodes for the derivatives to the computation graph



Gradients in Multilayer Neural Networks



- Numerical differentiation

  Forward sweep: $O(LN^2)$ operations

$$\frac{\partial E}{\partial \theta_{ln}} \approx \frac{E(\boldsymbol{\Theta} + \boldsymbol{\epsilon}_{ln}) - E(\boldsymbol{\Theta})}{\epsilon}, \qquad n = 1, \ldots N_l.$$

  - for each of the LN^2 parameters one forward sweep.
  - $O(LN^2 LN^2)$ operations; slow in deep networks, $L > 10^2, N > 10^6$

- Naive chain rule

$$\frac{\partial E}{\partial \boldsymbol{\theta}_l} = \frac{\partial E}{\partial \mathbf{h}_L} \frac{\partial \mathbf{h}_L}{\partial \mathbf{h}_{L-1}} \cdots \frac{\partial \mathbf{h}_{l+2}}{\partial \mathbf{h}_{l+1}} \frac{\partial \mathbf{h}_{l+1}}{\partial \mathbf{h}_l} \frac{\partial \mathbf{h}_l}{\partial \boldsymbol{\theta}_l}$$

  - for each $\boldsymbol{\theta}_l$ sweep back through chain of Jacobian-vector products
  - $O(LLN^2)$ operations

- Backpropagation (smart chain rule)

  - $O(LN^2)$ operations; only one forward and one backward sweep!

Backpropagation Algorithm

- Special case of more general **Automatic Differentiation** (AD) in reverse mode
- AD can be applied to arbitrary acyclic computation graphs
- Maximum O(N^2) operations for gradients for N nodes in graph
  (naive chain rule would be exponential in N)
- Operations for derivatives added to the computation graph
- Higher order derivatives are straightforward: derive the computation graph of the derivatives operations
- Some deep learning frameworks implement this “graph rewrite” method, e.g., **TensorFlow** and Theano

MNIST Example with Backpropagation and TensorFlow

- http://github.com/bbaeuml/lecture-ss17-deep-learning
- logistic_regression_tf.py
  - model is constructed and run as tensorflow computation graph
- logistic_regression_tf_gradient.py
  - use automatic differentiation to add subgraph for gradient of cost
- logistic_regression_tf_full.py
  - add node for weight update inside the graph
- logistic_regression_tf_optimizer.py
  - automatically add subgraph for complete gradient descent step